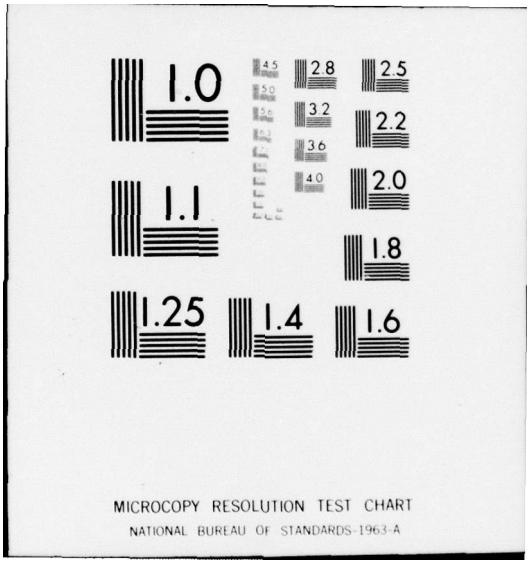
AD-A039 741 FEDERAL COBOL COMPILER TESTING SERVICE WASHINGTON D C F/G 9/2
TRANSFERABILITY OF FORTRAN BENCHMARKS,(U)
MAY 77 P OLIVER
UNCLASSIFIED FCCTS/TR-77/13 NL

1 OF 1
AD A039 741

END
DATE FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

DEPARTMENT OF THE NAVY

★ ★ ★ ★ ★

Automatic Data Processing
Equipment Selection Office

Washington, D.C.
20376

SOFTWARE DEVELOPMENT DIVISION

(4)

B.S.

TRANSFERABILITY OF FORTRAN BENCHMARKS

BIBLIOGRAPHIC DATA SHEET	1. Report No. FCCTS/TR-77/13	2.	3. Recipient's Accession No.
4. Title and Subtitle Transferability of FORTRAN Benchmarks			5. Report Date 9 May 1977 6.
7. Author(s) Paul Oliver			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Software Development Division ADPE Selection Office Department of the Navy Washington, D. C. 20376			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address ADPE Selection Office Department of the Navy Washington, D. C.. 20376			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

This Federal COBOL Compiler Testing Service (Software Development Division) monograph discusses policies and procedures designed to aid in the development of transferable FORTRAN benchmark programs.

17. Key Words and Document Analysis. 17a. Descriptors

Benchmarking
FORTRAN
Portability
Transferability
Conversion

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group 09/02

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	11
	20. Security Class (This Page) UNCLASSIFIED	22. Price

DEPARTMENT OF THE NAVY
AUTOMATIC DATA PROCESSING EQUIPMENT SELECTION OFFICE
WASHINGTON, D.C. 20376

16 January 1975

SDD TECHNICAL NOTE

Subj: Transferability of FORTRAN Benchmarks

Introduction

Although FORTRAN is a relatively simple language, its wide acceptance has resulted in a variety of extensions, and this proliferation of dialects greatly hampers the transferability of FORTRAN programs.

It must be said at the outset that complete transferability is probably not achievable, due to irreconcilable differences in machine architectures. Portable programs can, however, be produced with the aid of appropriate software tools. Whitten and de Maine give the following definitions for machine independent, configuration independent, and portable programs:

> "A source program is machine independent with respect to a set of computers if the program will compile, execute and produce the same results on each computer. A machine independent program is configuration independent if required computer resources can be dynamically allocated during program execution, and the amount of memory available to the program does not by itself determine the amount of data that can be processed. A source program which is both machine and configuration independent over a set of computer installations is said to be portable with respect to these installations."[1]

Portable FORTRAN Programs

The most systematic and comprehensive attempt at producing portable FORTRAN programs has been the development of PFORTRAN (Ref. 1). PFORTRAN is composed of four components:

1. A set of FORTRAN statements which is a common dialect of seven FORTRAN dialects and a superset of American Standard FORTRAN (Ref. 2). The seven dialects are:

CDC 6000 FORTRAN IV
HIS 6000 FORTRAN IV
IBM 360/370 FORTRAN G
RCA SPECTRA 70 FORTRAN
XDS Sigma 5/7 FORTRAN
UNIVAC 1108 FORTRAN V
PDP 11 FORTRAN IV - PLUS

The statements included in the set are:

ASSIGN	DIMENSION	GO TO
Assignment statements	DO	IF
BLOCK DATA	END	RETURN
CALL	ENTRY	Statement function
COMMON	EQUIVALENCE	STOP
CONTINUE	EXTERNAL	SUBROUTINE
DATA	FUNCTION	Type statement

2. The standard FORTRAN data types set has been extended to include kernels, bit strings, and virtual arrays. Bit strings are self-explanatory. Kernels are variable length data units, i.e., contiguous storage. Virtual arrays provide the means of maintaining configuration independence, since their size may exceed available memory.

3. A simple, machine-independent I/O function or interface.

4. A variable arithmetic precision package.

Unfortunately PFORTRAN is not itself portable, since a substantial portion of it consists of assembler language support subroutines. These could, of course, be re-coded. A second objection, purely from the standpoint of benchmark portability, is that PFORTRAN provides more than is really required for benchmarking. It is, therefore, questionable that the development of such a system for benchmark production is desirable.

It would certainly be feasible to develop a system analogous to the BPS (Ref. 3). The recognition and occasional conversion of non-standard FORTRAN statements would be a simple task. The absence of an even rudimentary data definition language in FORTRAN, and the overwhelming use of floating point arithmetic make the task of data file conversion formidable. Attempting the development of such a file conversion system is not advisable until some fundamental enhancements are made to FORTRAN.

FORTRAN Programming for Transferability

The ramifications of using extended or non-standard dialects in any language are well known. Unfortunately, most programming organizations tend to ignore the dangers of using non-standard language elements. Yet the only presently feasible way of producing transferable (machine-independent, not portable) FORTRAN benchmarks is to give careful consideration to unique system environment attributes, to avoid machine dependent coding practices, and to organize programs so as to maximize transferability. Most of the following material has been taken, largely unmodified, from Reference 4.

System Environment

It is impossible to control the machines/systems on which benchmark programs may be compiled. There are, however, several actions that can be taken to avoid system environment impact on transferability. The following two lists show attributes to consider in system changes and practices to avoid in general.

UNIQUE ATTRIBUTES TO CONSIDER

1. Word size, precision, and base.
2. Character size and the effect on core space and character oriented coding. Example:

   DATA MX/6CHARAC, 6HTER---/ versus

   DATA MX/4CHAR,4HACTE,4HR---/
3. Logical unit assignments.
4. Peripherals available (number and kind).
5. Code differences, e.g., EBCDIC, ASCII, etc.
6. Time controls.
7. Block and segment command words affixed to binary tape records.
8. Maximum and minimum record size constraints.
9. Printer and typewriter line size and character sets.
10. Incremental compiler restrictions (order of statements).
11. Internal representation (hex, octal, binary).
12. Executable memory size.

PRACTICES TO AVOID

1. Multiple file I/O.
2. Special disc or drum oriented instructions.
3. Literal unit assignments.
4. Special system routines such as sense switch options.

5. Typewriter I/O.

6. Use of hex and octal literals (internal representations).

7. Special characters (other than A-Z, 0-9, +, -, *, /, (,), ,, $ and ANSI standards).

8. Format controlled records of more than 120 characters.

9. Assembly language interfaces.

10. STOP and PAUSE (differing system response).

11. Assuming pre-initialized memory.

12. Character tests with machine dependent coding (input the character set).

13. Excessive use of labelled COMMON.

14. Word, byte, character, or sign implementation dependent coding.

Recommended Programming Practices

The ability to divide machine/system dependent and independent attributes into separate entities via sub-programs greatly enhances the transferability of programs. Furthermore, the ability to compile these entities individually aids in the speedy checkout of transferred modules. Only subroutines that are modified or corrected ever need be recompiled. In this way, machine dependent code may be economically isolated into separate modules.

Assuming that the isolation of system dependencies is feasible, the dependent segment should be separated into relatively homogenous sub-segments. This is necessary because some aspects will not be as dependent as others (formatted versus unformatted input/output, for example). Similarly, it may not be possible to completely purge an independent section of transfer problems. Thorough checks should be made for obscure problems connected with machine characteristics such as precision and round-off.

A typical program can be designed (divided) into the following modules.

1. Main control (independent)

   - Calls computational routines
   - Calls machine dependent routines
   - Calls main control loop
   - Calls independent exit routine
   - Capable of compilation on very basic system

2. Dependent initialization

   - Sense lights, overflow indicators, etc.
   - Character set (read in)
   - Format statements (read in)
   - Parameters (word size, character size, etc.)
   - Calls ERROR routine

3. Main control loop (independent)

   - Calls machine independent initialization
   - Defines blank COMMON
   - System independent tests and calculations to drive the job
   - Calls input routines
   - Calls ERROR routine
   - Calls output routine
   - Loop control
   - Calls general program
   - Returns to main control

4. Operating system interface (dependent)

   - Mechanism to exit system and dump
   - ERROR routine
   - Unformatted I/O routine (entered via computed GO TO)

5. Independent initialization

   - Usual program initialization process
   - Calls to operating system interface

6. ERROR routine (dependent)

   - Handles sense lights, overflow check, etc.
   - Handle recoverable errors (parity, end files, etc.)
   - Calls dump routines

7. Formatted input/output (independent)

   - All formatted input/output
   - Controlled by computer GO TO

8. General program (independent)

   - All calculations, data handling and program logic not present in separate modules
   - Calls dependent routines

Specification of machine/system dependent aspects by way of coded or card input parameters would aid program transferability. Unfortunately, many of the features such as word size, byte size, core size, record length and type, are deeply imbedded in individual compiler designs or have simply been overlooked. For instance, the number of characters per word is inherent in hollerity definitions such as DATA/X/6HABC DEF/. Similarly, there is no easy way to reduce the dimensions of arrays in COMMON. Of course, it has always been good programming technique to parameterize such things as DO loop indices or logical unit assignments, but much more is needed in this area.

Code Restraints

Many commonly utilized techniques in FORTRAN programs are severely detrimental to program portability because of machine/implementation differences inherent to individual compilers. The following list is a sample:

CODING TECHNIQUES WITH
UNPREDICTABLE VARIATIONS

1. Testing for 0. Example: IF(X.EQ.0.0) GO TO 10.
2. One terminal statement for a nested DO loop exit.
3. Termination of a DO with a complex statement such as an IF.
4. Literal strings of excessive length.
5. Altering a DO parameter within the loop.
6. Ambiguous statements such as

   N=N+FUNC(N)*N where FUNC(N) alter N.
7. Double exponentiation: A**B**C.
8. Assumption that some finite value will be substituted for a division by 0.
9. Recursive subroutines.
10. Variable length argument strings in subroutine CALLS.
11. Scattered specification statements.
12. Shifting by multiplication.
13. Using large numbers as DO indices. Example:

    DO 10 I=1, N where $N = 2^{17}$.

14. Assuming a loop is always executed once. Example:

    DO 10 I=4, N where N = 3.

15. Transferring into a DO range.

16. Assuming the value of an index outside a DO loop.

17. Returning from a subprogram via an assigned GO TO.

18. Assuming an incorrect computed GO TO variable will result in a default condition such as "falling through".

General Guidelines

1. Avoid usage of extensions to the Standard whenever possible.

2. Document extensively all dialect variations and machine/system dependent code, functions, general design, etc.

3. Modularize programs into machine/system dependent and independent sections.

4. Avoid assembly language code interfaces.

5. Do not use programming tricks dependent on machine idiosyncracies.

6. Design magnetic tape outputs for general compatibility. Avoid complicated blocking or binary (nonformatted) final outputs.

7. Lay out blank COMMON in one central routine and treat variables there as if they were global. This avoids duplication and is a form of documentation.

8. Always initialize memory even if the system on which the benchmarks are produced does it automatically.

9. Do not make programs dependent on the internal character representation of a particular machine.

10. Avoid operator interaction (typewriter I/O).

11. Estimate the range of data values and document same. The precision of integer and floating arithmetic is machine and software dependent.

Checklist for the Target Machine

1. Is the machine's core size adequate?

2. Are all intrinsics available?

3. Are all library routines available?

4. Is the FORTRAN standard?

5. Are word and byte size the same?

6. Is the precision adequate?

7. Is internal representation the same? (Octal, hex, binary, etc.)

9. Is the character set the same? Must card decks be converted?

10. Does operating system interface necessitate changes in the code? (i.e., are the systems CALLs the same?)

11. Can assembly language routines be rewritten easily?

12. Can the machine characteristics be defined in an initializing subroutine?

13. How will machine speed affect the programs? Must some be rewritten for efficiency?

14. Are all necessary FORTRAN statements available?

15. Does the machine/system dependence/independence modularity still hold?

16. What about data? Can some be used without reformatting? Do tapes have to be converted?

17. Will possible record size changes cause any storage problems?

18. Can rapid change and checkout be made? If not, why not?

19. Recompile independent modules first. Any errors which are encountered may signify some basic problems.

So that there is no misunderstanding about code restraint implications listed previously, the following justifications are included:

1. Due to inherent inaccuracies of floating point representations, it may happen that a test for a specific number will fail when actually it should pass. For example:

   A = 1. - 2.0/2.0 may come out to be .0000001 and a subsequent test for (A.EQ.0.) would fail.

2. Some compilers put special restrictions on nested DO loops that terminate on one statement. XDS Sigma 5/7, for instance, only allows the innermost DO to transfer directly to its termination point.

3. Compilers which allow termination of DO loops on IF statements disclaim the predictability of results (XDS Sigma).

4. Compilers may have inherent table size restrictions on character string length (CDC).

5. Altering a DO parameter within a loop may produce varying results depending on the mechanization of the DO (pre-test, post-test, index in core, index in register, etc.).

6. Ambiguous statements such as

   N=N+FUNC(N)*N

   depend on the mechanization of the scanning algorithm. If the original value of N is not stored in another temporary location FUNC(N) may destroy it.

7. Double exponentiation A**B**C is also not defined in the Standard and is dependent on the mechanized algorithm

   $(A^B)^C$ OR $(A)^{B^C}$

8. Division by 0 may result in values 0, 10^{39}, etc. depending on the system.

9. Most compilers do not allow recursive subroutines.

10. Some compilers will not deal successfully with missing arguments (variable length) in a subroutine CALL (mechanized at the RETURN and assumes the number of arguments is fixed).

11. Incremental compilers cannot handle scattered type, dimension, and DATA statements. Yet, this form of compiler is desirable from a speed/user interface standpoint.

12. Shifting algorithms are word-size and hardware (multiplication, sign representation) dependent.

13. Compilers may implement DO loops in index registers (IBM 7090) and hence set an upper limit on index values.

14. Loops of the form DO 10 I=K, J where J K may or may not be executed once depending on where the test for completion is made.

15. Most compilers do not guarantee the results of transfers into a DO loop.

16. The addressing scheme of some machines/compilers allows for passing of statement label addresses for an assigned GO TO to subroutines.

17. Some compilers generate code for testing the computed GO TO arguments. The last statement label or next statement becomes the error condition default.

Conclusion

The development of a benchmark sanitization system for FORTRAN is not recommended at this time. What could be done easily (instruction conversion) is best handled through code constraints. Data conversion and the resolution of unique machine characteristics (word size) is not feasible until a significant data definition capability (at least as comprehensive as PL/I's, if not COBOL) is added to FORTRAN.

The suggestions made in this note should, if followed, be sufficient for the production of highly portable FORTRAN benchmark programs.

PAUL OLIVER
Director, Software Development

REFERENCES

1. "A Machine and Configuration Independent FORTRAN: Portable FORTRAN (PFORTRAN)," D. E. Whitten and P. A. D. de Maine, Computer Science Department, Pennsylvania State University (1974), University Park, Pennsylvania, 16802.

2. "American Standard FORTRAN," American Standards Association (1966).

3. "System for Efficient Program Portability," G. N. Baird and L. A. Johnson, Proc. NCC 74, AFIPS Press.

4. "Programming for Transferability," Joel E. Heiss, et al, International Computer Systems, Inc. (1972), 10801 National Boulevard, Los Angeles, California, 90064.